NRL Memorandum Report 6493

SMMS Software Module Guide

MARK R. CORNWELL

Center for Secure Information Technology Branch
Information Technology Division

July 28, 1989

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Memorandum Report 6493	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Research Laboratory; 6b OFFICE SYMBOL (If applicable) Code 5540	7a NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Washington, DC 20375-5000	7b ADDRESS (City, State and ZIP Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION Space & Naval Warfare Sys Cmd; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Washington, DC 20363-5100	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
35167G		68003	

11 TITLE (Include Security Classification)
SMMS Software Module Guide

12 PERSONAL AUTHOR(S)
Cornwell, M.R.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Interim	FROM 1/86 TO 8/88	1989 July 28	24

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software engineering; Software design
			Trusted systems; Computer security
			Software documentation; Message systems KK)

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This document describes the module decomposition for the Secure Military Message System (SMMS) full scale prototype software being produced at the Naval Research Laboratory. It provides an orientation for software engineers who are new to the SMMS, explains the principles used to design the structure, and shows how responsibilities are allocated among the major modules.

The guide is intended to lead a reader to a module that implements a particular aspect on the system. It states the criteria used to assign a particular responsibility to a module and arranges the modules in such a way that the reader can find the information relevant to this purpose without searching through unrelated documentation.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL Mark R. Cornwell	22b TELEPHONE (Include Area Code) (202) 767-6698	22c OFFICE SYMBOL Code 5540

DD Form 1473, JUN 86 Previous editions are obsolete SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

CONTENTS

SMMS SOFTWARE MODULE GUIDE

I. INTRODUCTION

PURPOSE

The Secure Military Message System (SMMS) Software Module Guide describes the module structure of the secure military message systems software produced by the Naval Research Laboratory. It provides an orientation for software engineers who are new to the SMMS, explains the principles used to design the structure, and shows how responsibilities are allocated among the major modules.

This guide is intended to lead a reader to the module that implements a particular aspect of the system. It states the criteria used to assign a particular responsibility to a module and arranges the modules in such a way that a reader can find the information relevant to this purpose without searching through unrelated documentation.

The module guide should be read before any other design documentation for the SMMS software, because the guide defines the scope and contents of the individual design documents.

This guide describes and prescribes the module structure. Changes in the structure will be promulgated as changes to this document. Changes are not official until they appear in that form. This guide is a rationalization of the structure, not a description of the design process that led to it.

PROGRAM FAMILIES

The term ***program family*** applies to related programs in a way analogous to how the term hardware family applies to related computer hardware. The concept of a hardware family was introduced with the IBM/360 family of computers and is now well established. The IBM/360 family consists of a number of computer hardware systems that share a common set of design decisions (mainly concerning the instruction set) but differ in ways (cost, performance, physical dimensions, hardware technology) that allow different family members to fulfill different needs.

Two programs are said to be part of the same *family* if it is beneficial to consider their similarities before considering their differences [1]. Similar software applications may offer different enough requirements that the same program will not satisfy the needs of both. However, the requirements might be satisfiable by two different programs that share many common design decisions and much of the same code. An approach to system design that strives to take advantage of these similarities by making common design decisions early and striving to maximize commonality between the different programs is termed a family approach to designing software. This family approach should reduce costs by avoiding duplication of effort over the lifecycle of the software. Effort spent on designing, programming, maintaining, and enhancing a family of similar systems should be less than that required to perform the same services on independently developed systems.

The SMMS Family is composed of a number of different systems that share a core of common design decisions. Among the common design decisions is the module structure. This common module structure should contribute to our ability to share code, design effort, and security analysis effort across family members. The structure described in this guide provides a common module decomposition used by all the systems in the SMMS family.

PREREQUISITE KNOWLEDGE

Readers are assumed to be familiar with the terminology and organization of *Software Requirements for the Secure Military Message System Family* [2], which will be referred to as "the requirements document"; the first chapter of the requirements document, in particular, will be referred as "the security model"; and with *Security Architecture for a Secure Military Message System* [3], which will be referred to as "the security architecture". They should have a general idea of the functions performed by secure military message system software, know something about electronic mail, message processing, and computer security models. These are described in adequate detail in the above documents.

Manuscript approved October 26, 1988.

ORGANIZATION

Section II gives the background for the design. It states 1) the goals that motivated the module design decisions presented in this document; 2) the basic principles on which the design is based; and 3) the relationship between the module structure and other structures of the SMMS software.

Section III, the main body of the document, presents a hierarchical decomposition of the software into top-level, second-level, and third-level modules. The modules at each level are components of modules of the next higher level.

Terms, such as "modules", that are used with a special meaning in NRL's Software Cost Reduction (SCR) methodology, are defined in the glossary. Readers who are not familiar with the SCR terminology should study the glossary before reading further.

ACKNOWLEDGEMENT

This module guide is an adaptation of the A-7E Software Module Guide (NRL Memorandum Report 4702) by K.H. Britton and D.L. Parnas [10]. Much of the introductory material is taken verbatim with the minor modification of substituting "SMMS family" for "A-7E". This was possible because that material describes general software engineering principles that we have chosen to adopt for the SMMS. The top-level module decomposition of the A-7E design has been maintained in the SMMS design. The decomposition at lower levels differs and is more specific to the SMMS.

II. BACKGROUND

THE SMMS SOFTWARE STRUCTURES

A structural description of a software system shows the program's decomposition into components and the relations between those components. SMMS programmers must be concerned with four structures: (a) the module structure, (b) the *uses* structure, (c) the process structure, and (d) the domain structure. This section contrasts these structures.

(a) A module is a work assignment for a programmer or programmer team. Each module consists of a group of closely related programs. The *module structure* is the decomposition of the program into modules and the assumptions that the team responsible for each module is allowed to make about the other modules.

(b) In the *uses structure* the components are programs, i.e., not modules but parts of modules; the relation is "requires the presence of". We say one program *uses* another program if the former requires the presence of the latter in order to meet its specification. The uses structure determines the executable subsets of the software [4]. Guidelines for the design of the SMMS *uses* structure are given in [5].

(c) The *process structure* is the decomposition of the run-time activities of the system into units known as processes. A process is a subset of the run-time events of the system used as administrative units in the run-time allocation of processors. Processes are not programs; there is no simple relation between modules and processes. The implementation of some modules may include one or more processes, and any process may invoke programs in several modules. Guidelines for the SMMS process design are given in [6].

(d) The *domain structure* is the decomposition of access privileges to data and programs into sets known as domains. At any instant, each process is associated with a domain that determines what run time activities may be part of that process. For example, one domain may allow a process to alter the security markings of objects while another may not. The domain structure of the SMMS is described in [3].

The rest of this document describes the module structure.

GOALS OF THE SMMS MODULE STRUCTURE

The decomposition into modules has two overall goals. The first goal is to increase security assurance by allowing those modules that impact the security of the system to be isolated and analyzed. The second goal is reduction of software cost by allowing modules to be designed and revised independently. This goal also includes reduction in cost for the family by allowing different family members to be generated by substituting, adding, or deleting modules.

Specific security assurance goals of the module decomposition are:

(a) The responsibility of each module for maintaining system security should be explicit.

(b) Each module should be simple enough that its security assertion can be verified.

Specific cost-reduction goals of the module decomposition are:

(c) It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behavior of other modules.

(d) The ease of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed. It should be possible to make likely changes without changing any module interfaces; less likely changes may involve interface changes, but only for modules that are small and not widely used. Only very unlikely changes should require changes in the interfaces of widely used modules.

(e) It should be possible to make a major software change as a set of independent changes. Programmers changing the individual modules should not need to communicate. If the interfaces of the modules are not revised, it should be possible to run and test any combination of old and new module versions.

As a consequence of the goals above, the SMMS software is composed of many small modules. They have been organized into a tree-structured hierarchy; each nonterminal node in the tree represents a module that is *composed of* the modules represented by its descendents. The hierarchy is intended to achieve the following additional goals:

(f) A software engineer should be able to understand the responsibility of a module without understanding the module's internal design.

(g) A reader with a well-defined concern should easily be able to identify the relevant modules without studying irrelevant modules. This implies that the reader be able to distinguish relevant modules from irrelevant modules without looking at their components.

DESIGN PRINCIPLES

The SMMS module structure is based on two decomposition criteria: proof-based decomposition [3] and information hiding [7]. The first criterion is concerned with security assurance, the second with ease of change.

A security proof forms the basis for generating the list of security critical properties, and for verifying that, taken together, these properties form a sufficient condition for the system to be secure. According to the proof-based methodology, those properties upon which the security of the system rests should be encapsulated in verifiable modules. A property of the system is encapsulated in a module M if it is impossible to invalidate that property by altering any module other than M. (One possible approach to showing that a property cannot be invalidated is to assume that the mechanisms supporting the domain structure function correctly and base arguments on that domain structure.) If a property is thus encapsulated, then it is possible to verify that the property holds for the system by inspecting only the module in which the property is encapsulated.

The security proof is documented in a separate document and is not described further in this guide. Refer to the security architecture and security proof documentation [3,11] for specific details about the proof.

According to the the information hiding principle, system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change. Every data structure is private to one module; its variables may be directly accessed by one or more programs within the module but not by programs outside the module. Any other program that requires information stored in a module's data structures must obtain it by calling that module's programs.

Applying this principle is not always easy. It is an attempt to minimize the expected cost of software and requires that the designer estimate the likelihood of changes. Such estimates are based on past experience, and may require knowledge of the application area, as well as an understanding of hardware and software technology.

In a few cases information that is likely to change must be communicated between modules. To reduce the cost of software changes, use of some modules or portions of a module interface, may be restricted. Restricted interfaces are indicated by "(R)" in the documentation. Often the existence of certain smaller modules is itself a secret of larger modules. In a few cases, we have mentioned such modules in this document in order to clearly specify where certain functions are performed. Those modules are referred to as hidden modules and indicated by "(H)" in the documentation.

MODULE DECOMPOSITION

Four ways to describe a module structure based on information-hiding are: (1) by the *roles* played by the individual modules in the overall system operation; (2) by the *secrets* associated with each module; (3) by the *facilities* provided by each module; and (4) by the *security properties* encapsulated in each module. This document describes the module structure by characterizing each module's secrets. Where useful, we also include a brief description of the role of the module. The description of facilities is relegated to the module specifications (e.g. [8]).

For some modules we find it useful to distinguish between a *primary secret*, which is hidden information that was specified to the software designer, and a *secondary secret* which refers to the implementation decisions made by the designer when implementing the module designed to hide the primary secret.

Although we have attempted to make the decomposition rules as precise as possible, the possibility of future changes in technology makes some of the boundaries fuzzy. Some sections point out fuzzy areas and discuss additional information used to resolve ambiguities.

SECURITY ASSERTIONS FOR MODULES

For those modules that have some security responsibility we provide a description of that responsibility. At this level of detail, the security responsibility is stated in general terms. It is intended that the module interface specifications make these security assertions more precise as the module interfaces are refined.

Separate documents provides the supporting evidence that we have satisfied the proof-based decomposition criteria. A *security architecture* provides an overview of the structures of the system as it relates to security and our rational for asserting the structures defined provide the required security. Stronger evidence will be provided by a security proof that derives from the formal security model a set of assertions that taken together make up a sufficient condition for the system to be secure.

Assurance must also be provided that the security assertions for the modules hold for the module implementations generated by the programmers. The security assertions are intended to be verifiable by inspection of the code and specifications. This inspection may include formal (mathematical) verification techniques. In some cases, mathematical techniques will not be applicable and assurance will have to rely on the strengths of the arguments provided by the implementors and the soundness of the reviewers' judgement.

MODULE INITIALIZATION

Every module in the SMMS software can contain variables that must be given initial values when the SMMS software is started up. Each module contains a program for initialization, that will be called when startup occurs. There will be a main initialization program that is invoked upon startup. It will invoke the initialization programs for the three top-level modules. The initialization program for a module will call the initialization programs for selected second-level modules.

The initialization program for a module at the leaf of the module tree is a part of that module. The initialization program for a higher level module is contained in a submodule of M, called the initialization module of M. These initialization modules will not be described further in this document.

FUTURE ADDITIONS TO MODULES

It is often the case that a particular version of the system may not need features of a module that are likely to be needed in other versions. Where we have identified features that may not be needed in the initial version, they are included in the module interface descriptions but it is noted that they will not be needed in the initial version. The programmer can use this information about likely future additions to design his software for easier extension [4].

TREATMENT OF UNDESIRED EVENTS

Development versions of all modules will check for and report undesired events (UEs). Each module interface description contains a list of possible UEs. In general, such a list should constitute a classification of all the things that might go wrong. It should include hardware errors, software errors and errors caused by the using program. However, some of the UE detection and correction must be removed from the production version to improve performance. In the SMMS context, module specifications will say what UE detection and correction may and may not be removed. For a more complete discussion of UEs see [9].

Attempted security violations will normally be treated as UE's. Attempted security violations include such events as a user trying to display a message on a terminal whose classification does not dominate that of the message, or a user without proper authorization trying to release a message.

The remainder of this report provides a top-down overview of the module structure.

III. SMMS MODULE STRUCTURE

A: TOP-LEVEL DECOMPOSITION

The software system consists of the three modules described below.

A:1 HARDWARE-HIDING MODULE

The Hardware-Hiding Module includes the programs that need to be changed if any part of the hardware or its operating system software is replaced by a new unit with a different hardware/software interface but with the same general capabilities. Any likely replacement of hardware or operating system software should *not* require changes to the interface to this module. This module implements a virtual hardware and operating system base that is used by the rest of the SMMS software.

A:2 BEHAVIOR HIDING MODULE

The Behavior-Hiding Module includes programs that need to be changed if there are changes in the user-visible behavior of the system. Users may be human users or devices and software that interface with the SMMS. Behavior includes both required behavior prescribed by the requirements document and security model as well as translation of generally prescribed behavior of the intermediate command language to a specific human interface. (The intermediate command language is an abstraction of the user command language and is intended to capture the required semantics of the user commands without prescribing their syntax or other characteristics of the human interface.) These programs determine the values to be sent to the virtual output devices provided by the Hardware-Hiding Module.

A:3 SOFTWARE DECISION MODULE

The Software Decision Module hides software design decisions that are based upon mathematical theorems, physical facts, and programming considerations such as algorithmic efficiency and accuracy. The secrets of this module are not described in the requirements documents. This module differs from the other modules in that both the secrets and the interfaces are determined by the software designers. Changes in these modules are more likely to be motivated by a desire to improve performance than by externally imposed changes.

B: SECOND LEVEL DECOMPOSITION

B:1 HARDWARE-HIDING MODULE DECOMPOSITION

The Hardware-Hiding Module comprises two modules.

B:1.1 ABSTRACT BASE MACHINE MODULE

The Abstract Base Machine Module hides characteristics of the hardware/software interface that are likely to change if the computer is changed or replaced: number of processors, instruction set, capacity for concurrent operations, operating system. This module may be an off-the-shelf compiler plus a standard set of library routines for managing concurrency.

The security assertion for the Abstract Base Machine is basically that the entire module operate correctly according to its specification. In particular, the protection mechanisms of the abstract base machine must be trustworthy because they will be the mechanisms from which protection mechanisms specific to message system security will be constructed.

B:1.2 DEVICE INTERFACE MODULE

The Device Interface Module hides characteristics of peripheral devices considered likely to change.

For secure operation, each device interface must implement the protocol used to control the device correctly. Under normal operation it should provide that all the data the device communicates to the system software is provided without any deletions, additions or alterations. When a device malfunctions, the device interface must be able to sense that fact and communicate it to appropriate software so that appropriate action can be taken to ensure secure operation.

Notes on the Hardware-Hiding Module Decomposition

Our decomposition has it roots in the distinction commonly made between a computer and its peripheral devices. Peripheral devices are commonly physically separated from the processor and serve as an interface to human users or other computer systems. The SMMS requirements documentation [2] makes this distinction with respect to input/output devices.

A particular peripheral device will be represented in the Device Interface Module if the SMMS software must be aware of its characteristics in order to perform its function. Unless the SMMS system makes some specific assumptions about the device it will not be considered part of the Device Interface Module.

It is likely that there will be physical devices that are totally invisible to the SMMS software. An example is a disk drive upon which the "virtual hardware" maintains it file system. In so far as these devices are not necessary at the Abstract Base Machine interface for the implementation of the SMMS they will not be considered part of the Device Interface Module. Be careful not to confuse this common distinction between the computer and its devices with the specific distinction defined here.

B:2 BEHAVIOR-HIDING MODULE DECOMPOSITION

The Behavior-Hiding Module consists of three modules.

B:2.1 INTERMEDIATE COMMAND LANGUAGE MODULE

The Intermediate Command Language Module hides the Intermediate Command Language (ICL) from the rest of the system. This includes the number of commands, their names, their parameters, and the semantics of the commands. It should be possible to add new commands, and delete old ones or change their meanings by changing only programs in this module. This module allows different family members to include only the ICL programs supporting a member's ICL subset without expending resources to support capabilities outside its subset.

The programs that implement the ICL commands must faithfully represent the user-level objects (messages, message files, directories, etc.) using primitives provided by the entity monitor level (entities, containment relationships) in a way consistent with the interpretation of the security model for the message system.

B:2.2 HUMAN INTERFACE MODULE

The Human Interface Module hides the syntactic level of the user interface. This includes how actions performed by the user (typing characters, moving pointing devices, etc.) are interpreted to construct ICL requests. It also includes how the outputs of ICL commands are presented to the user. Note that this module does not assume the names of ICL commands, their number, nor the number and types of parameters they expect. The human interface finds out this information by using the ICL module at system generation time or later.

The security assertions for this module are 1) that the user-level actions are correctly translated into requests in the intermediate command language, 2) that requests are transmitted to the rest of the system in the order they were made, 3) that no additional requests are added, and 4) that no requests are deleted.

B:2.3 SECURITY POLICY MODULE

The primary secret of this module is the SMMS security policy; its secondary secret is how the mechanisms of the abstract base machine are used to implement secure entities and how these mechanisms assure the rest of the system that the security rules are enforced. It is intended that the major part of the properties that need to be proved about the system for security assurance will be provable by reasoning about the implementations of programs in this module.

The Security Policy Module is responsible for providing a reference monitor that isolates the security relevant state of the system and ensuring that the state transitions the system goes through are as the security model prescribes. See the entity monitor module for more details.

B:2.4 EXTERNAL FORMATS MODULE

The External Formats Module hides message and audit data formats imposed by the requirements of the message system's interfaces to other systems. How to determine the addressees of a message, how to determine what fields a message contains and how to extract or set the contents of message fields are the primary secrets of this module.

This module must correctly implement its specification and be true to the standard external message formats. Some parts of the format can be considered especially critical such as the security labeling and releaser information.

B:3 SOFTWARE DECISION MODULE DECOMPOSITION

The Software Decision Module hides choices made strictly by software designers, not dictated by the requ[illegible]ts document or the security model.

B:3.1 SOF[illegible]RE UTILITY MODULE

The Software Utility Module hides implementations of programs of general usefulness to programmers programming different parts of the system. Programmers will be required to submit programs that are of possible general utility to those implementing other parts of the system for inclusion in this module.

The security assertion for this module will depend on the programs that eventually end up in this module.

B:3.2 SYSTEM GENERATION MODULE

The System Generation Module hides values of system parameters not bound until system generation time, which versions of modules are included during system generation, and which family member is being constructed.

The security assertion for this module is that the features used in generating the SMMS system must operate as specified. All parameters that are required by the modules must be provided and meet constraints set by the module specifications (such as bounds, type, etc.)

C: THIRD-LEVEL DECOMPOSITION

C:1 HARDWARE-HIDING MODULE

C:1.1 ABSTRACT BASE MACHINE MODULE DECOMPOSITION

C:1.1.1 FILE SYSTEM MODULE (R)

The File System Module hides the conventions used to access programs and data residing on secondary storage. It hides the mechanisms provided by the underlying machine and operating system to restrict access (e.g. reading, writing, executing) to programs and data residing on secondary storage.

C:1.1.2 DOMAIN DEFINITION MODULE

The Domain Definition Module hides the mechanisms for implementing domains, the mechanisms for controlling the entry points to a domain, and the algorithms for modifying domains and switching between domains.

C:1.1.3 EXCLUSION REGIONS MODULE

The Exclusion Regions Module hides the mechanism used to implement the exclusion relation on regions of code. A region of code is a set of source code statements. A region of code is said to exclude another if it is not safe to allow execution to enter the latter when execution is within the former.

C:1.1.4 PROCESS CONTROL MODULE

The Process Control Module hides the number of processors and how processes are allocated among them, the data structures and operations required to load a process on a processor, and the data structures required to represent processes and keep track of their current states.

C:1.1.5 PROCESS SYNCHRONIZATION MODULE

The Process Synchronization Module hides the data structures and algorithms used for the synchronization of processes and how synchronization operations are made indivisible.

C:1.1.6 PROGRAMMING LANGUAGE MODULE

The Programming Language Module hides the data representation and data manipulation and sequence control instructions supported by the base machine.

C:1.1.7 SECURITY LEVEL MODULE

The Security Level Module hides the implementation of the abstract data type that represents security levels, and also the implementation of the operations provided to manipulate objects of that type.

C:1.2 DEVICE INTERFACE MODULE DECOMPOSITION

C:1.2.1 TERMINAL INTERFACE MODULE

The Terminal Interface Module hides characteristics of terminals such as screen size, keys available, control sequences, character encodings.

C:1.2.2 PRINTER INTERFACE MODULE

The Printer Interface Module hides characteristics of hard-copy output devices, such as printers, typesetters, etc.

C:1.2.3 READER INTERFACE MODULE

The Reader Interface Module hides characteristics of read-only input devices, such as optical character readers.

Documents must be read correctly and in their entirety without any additions or deletions.

C:1.2.4 NETWORK INTERFACE MODULE

The Network Interface Module hides characteristics of network interfaces to communications networks, local area networks, etc.

The protocols for communication on the external network must be implemented correctly, especially with respect to addressing, labeling information, and signaling the boundaries on labeled information such as messages.

C:2 BEHAVIOR-HIDING MODULE DECOMPOSITION

C:2.1 ICL MODULE DECOMPOSITION

The Intermediate Command Language Module consists of three modules.

C:2.1.1 COMMAND PROCESSING MODULE

The Command Processing Module hides the ICL command set and information concerning the exact number and types of parameters required. It provides a generalized interface for invoking any possible ICL command, without reference to the exact syntax of that invocation.

Its security responsibility is that it use the SP.EM.ICL Control Module in the fashion required to correctly invoke the ICL operation programs on the ICL Commands Module interface.

C:2.1.2 ENTITY DATA TYPES MODULE (H)

The Entity Data Types Module hides the representation of the entity data types that are acted upon by the ICL commands. These data types correspond to abstract views of user-level objects such as messages, message files, forms, etc. The operations provided here make finer grained accesses to ICL objects than do the ICL commands.

The submodules of the Entity Data Types Module have been divided up to each contain access programs that operate on a particular entity data type. They each have a primary secret that is the representation of the data type and how it is manipulated by the access programs. Access programs on each of these module's interfaces will be able to directly affect (alter or inspect) only items of the data type known to that module. Operations in one submodule will be able to indirectly affect objects of other types through access programs on the other submodule interfaces.

The Entity Data Types Module contains the following submodules:

General Operations
Comment Operations
Device Operations
Directory Operations
Filter Operations
Field Operations
Form Operations
Message Entry Operations

Message File Operations
Message Operations
Paragraph Operations
Profile Operations
Text Operations
Utility Operations

The security responsibility of this module is that it implement the abstract ICL level data types in terms of the security model objects and functions provided by the SP.Entity Monitor, and do so in a way consistent with the security model interpretation for the message system.

C:2.1.3 ICL COMMANDS MODULE (H)

The ICL Commands Module provides individual programs for each of the ICL commands listed in the requirements document [2]. ICL commands may affect more than one data item or type at a time. For example, the ICL command to send a message can both alter the sender's message entry and the inboxes of the recipients. Its primary secret is the semantics of the ICL commands.

The ICL Commands Module contains the following submodules, which have been divided along the lines of the categories of commands found in [2]:

Device Commands Module
Directory Commands Module
Filter Commands Module
Form Commands Module
Message File Commands Module
Message Commands Module
Profile Commands Module
System File Commands Module
Text File Commands Module
User Commands Module

Its security responsibility is that it use the types defined in the Entity Data Types module in a way faithful to the interpretation of those data types to implement the ICL commands.

C:2.2 HUMAN INTERFACE MODULE DECOMPOSITION

The Human Interface Module is not decomposed into further modules at this time. This is because different implementations of this module may meet vastly differing requirements and must necessarily differ greatly in their internal structure. Another reason is that we want to be able to integrate existing user interface programs with the SMMS. Future revisions to this guide may decompose this module further.

C:2.3 SECURITY POLICY MODULE DECOMPOSITION

The Security Policy Module contains two submodules. Other modules may be defined in the future.

C:2.3.1 ENTITY MONITOR MODULE

The primary secret of the Entity Monitor Module is the security policy of the SMMS, making this module responsible for completely isolating the security relevant part of the system state. This module acts as a monitor for secure entities and other abstractions recognized by the security model. It is responsible for detecting attempted security violations and preventing them as well as protecting itself from outside tampering. It also hides the classifications of entities and the clearances of users. The specification must be consistent with the formal security model for the SMMS family.

C:2.3.2 MESSAGE ENTITY CONVERSION MODULE

The Message Entity Conversion Module hides the mapping between the abstract message structure provided by the message formats module and messages represented using entity structures of the entity monitor module.

Its security responsibility is to correctly translate the security relevant part of the message data and correctly construct the corresponding entity. The most critical part of this process is the security labeling information, the addressees, the hierarchical relationship of the parts of the message, and correctly delimiting the beginning and ending of the messages.

C:2.4 EXTERNAL FORMATS MODULE DECOMPOSITION

C:2.4.1 MESSAGE FORMATS MODULE

The Message Formats Module hides the external formats of formal military messages.

C:2.4.2 MESSAGE ACCOUNTABILITY LOGS MODULE

The Message Accountability Logs Module hides the format of the message accountability logs.

C:3 SOFTWARE DECISION MODULE DECOMPOSITION

The Software Decision Module consists of two modules.

C:3.1 SOFTWARE UTILITY MODULE DECOMPOSITION

This module may be decomposed as the number of programs in it grows. It currently contains no programs.

C:3.2 SYSTEM GENERATION MODULE DECOMPOSITION

C:3.2.1 SYSTEM GENERATION PARAMETER MODULE

This module provides values for all the system generation parameters defined in other modules, including those specified in module interfaces and those defined in the module implementations. There is a submodule of the System Generation Parameter Module for each module in the rest of the system; each of these submodules is in turn composed of an external parameter submodule and an internal parameter submodule. External parameters of a module are available to other modules; internal parameters are secrets of the module. The primary secrets of this module are the values of the parameters for a particular version of the system.

C:3.2.2 MODULE VERSION SELECTION MODULE

This module stores alternative implementations of each module. It allows a user to indicate which alternative(s) should be chosen for each module. If the module implements an abstract data type, a different alternative may be specified for each variable of that type. The secret of this module is the library structure used to store the various versions of the modules and the procedure for inserting the selected implementation in the code.

C:3.2.3 SUBSET SELECTION MODULE

This module selects subsets of each module in order to assemble a desired subset version of the system. Its primary secret is the "Uses" relation; the secondary secrets are the representation of the relation and the algorithms used to select the programs that will be included in the resulting system.

C:3.2.4 SUPPORT SOFTWARE MODULE

This module includes additional software required to generate and check out a running system, including a macro-processor, a testing program and test data (either stored data or tools to automatically generate it), and communications software. The compiler resides in the Abstract Base Machine Module and not in this module because it knows the secret of the base machine's instruction set.

This module must operate according to its specification to the extent that its features are depended upon by the SMMS system.

IV. REFERENCES

[1] Parnas, D.L., "On the Design and Development of Program Families," *IEEE Trans. on Software Eng.*, Vol. SE-2, pp.1-9, Mar. 1976.

[2] Quinn, J.T., C.L. Heitmeyer, M.R. Cornwell, C.E. Landwehr, J.D. McLean, *Software Requirements for the Secure Military Message System Family*, NRL Memorandum Report # pending.

[3] Cornwell, M., Moore, A., Security Architecture for a Secure Military Message Systems, NRL Formal Report 9187, April 28, 1989.

[4] Parnas, D.L, "Designing Software for Extension and Contraction", *Proceedings of the 3rd International Conference on Software Engineering* (10-12 May 1978), pp. 264-277.

[5] Cornwell, M.R., *SMMS FSP "Uses" Specification*, NRL Technical Memorandum 5590-74, 4 March, 1988.

[6] Process structure documentation, to be published.

[7] Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules", *Comm. ACM*, Vol. 15, No. 12 (December 1972), pp. 1053-1058.

[8] Cornwell, M.R., Entity Monitor Module Specification (DRAFT)

[9] Parnas, D.L., H. Wuerges, "Response to Undesired Events in Software Systems", *Proc. Second Int. Conf. Software Eng.*, pp. 437-446, 1976.

[10] Britton, K. and D.L. Parnas, *A-7E Software Module Guide*, NRL Memorandum Report 4702, Dec. 8, 1981.

[11] Cornwell, M., A Software Engineering Approach to Designing Trustworthy Software IEEE Proceedings, Computer Society Symposium on Security and Privacy, pg. 148-156, May 1989.

V. GLOSSARY

abstract interface	an abstraction that represents more than one interface (see interface, module interface); it consists of the assumptions that are included in all of the interfaces that it represents.
abstraction	a description of a set of objects that applies equally well to any one of them. Each object is an instance of the abstraction.
access function	see *access program*
access program	a program that may be called by programs outside of the module to which it belongs. Most run-time communication between modules is effected by invocation of access programs. There are several different sorts of access functions: some return information to the caller, some change the state of the module to which they belong, and some do both.
domain	a set of access permissions. The access permissions specify a set of objects and set of operations that may be applied to those objects.
hidden submodule	a submodule whose existence is part of the secret of the parent module.
interface	(1) between two programs: the assumptions that each programmer needs to make about the other program in order to demonstrate the correctness of his own program. (2) between a program and a device: assumptions about the device that must be accounted for in the program in order for the program to work as expected.
internal program	a program that is not accessible to programs outside the module; the existence of the internal program is part of the secret of the module.
module	a programming work assignment consisting of one or more programs. A module may be divided into smaller modules (submodules).
module facility	the access programs and events provided by a module in order to allow user programs to be independent of the module secret. A complete description of a facility is a specification of the module.
module hierarchy	a hierarchy defined by the relation "contains" on pairs of modules.
module implementation	the algorithms, data structures, and programs that satisfy the module specification.
module interface	the set of assumptions that the authors of external programs may make about the module. It includes restrictions on the way that the module may be used. In the SMMS software, modules communicate either by one module using access program from the other module, or by one module inspecting the value of a shared object that was set by the other module. The interface consists of assumptions about the availability of the access programs, the syntax of the calls on the access programs, the behavior of the access programs, and the meaning of events. See also interface.

module secret	see secret, module implementation
module specification	a description of as module interface; see also module facility.
module's structure	the way that a software module is divided into submodules and programs
primary secret	the characteristics other than decisions by the module designer that a module is intended to hide. See also secondary secret
process	a subset of the run-time events of the system used as administrative units in the run-time allocation of processes.
process definition	the program that controls the sequence of actions by a process.
program	a named, machine executable, description of an algorithm. The name may be used to invoke the program's execution. A program may include a description (declaration) of the data structures that it uses; it may invoke other programs and refer to data structures that have been described in other programs. See also subprogram, process definition.
secondary secret	software design decisions made to implement the abstractions that hides the primary secret.
secret	all the facts about the module that are **not** included in its interface; i.e., assumptions that user programs are not allowed to make about the module. The correctness of programs in other modules must not depend on those facts. The secrets tell how the module's specification has been satisfied. See also module specification, primary secret, secondary secret.
security property	A property from among a set of properties that taken together are a sufficient condition for the system to be deemed secure.
submodule	any module that is a component of a higher level module.
subprogram	a subprogram is a program that can be invoked by another program. A subprogram may be either a subroutine or macro.
sysgen parameters	a symbol used as a placeholder for values that will be supplied just before the system is generated.
undesired event (UE)	a run-time event that the designers hope will not occur. Production versions of the SMMS will assume that a significant portion of them do not occur.
undesired event assumption	assumptions about what constitutes improper use of a module by user programs, e.g., calling an access program with parameters of the wrong type.
use, uses	Program A uses program B is there must be a correct version of B present for A to run correctly. A program uses a module if it uses at least one program from that module. A module uses another module if at least one program uses that module.
user programs	all programs that use programs from a module but are not part of that module. The term "user" is relative to the module being discussed.
virtual computer	a computer-like set of instruction implemented, at least in part, by software.
virtual machine	see virtual computer

visible submodules	submodules whose existence is visible to user programs.